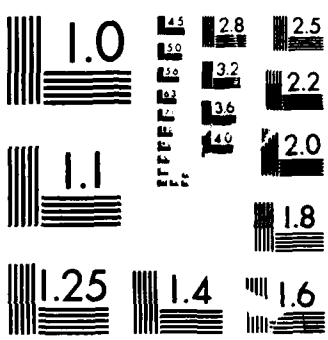
AD-A187 559 THEORY AND PRACTICE OF FAULT TOLERANCE IN DISTRIBUTED SYSTEMS(U) TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES K M CHANDY ET AL 30 MAR 87 AFOSR-TR-87-1462 1/1

UNCLASSIFIED AFOSR-85-0252 F/G 12/5 NL



AFOSR-TR- 87-1462

(2)

Air Force Scientific Report

AFOSR 85-0252

6/15/85 through 10/14/86

K. M. Chandy and J. Misra

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
(512) 471-4353

March 30, 1987

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
AFOSR/NM Bolling AFB DC 20332-6448	61102F	2304	A3	

11. TITLE (Include Security Classification)

Theory and Practice of Fault Tolerance in Distributed Systems

12. PERSONAL AUTHOR(S)

Prof. K.M. Chandy & Prof. J. Misra

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 6/15/85 TO 10/14/86	March 30, 1987	

16. SUPPLEMENTARY NOTATION

17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

RESEARCH OBJECTIVES

Over the years 1981-1985, our research on distributed systems, supported by AFOSR 81-0205, resulted in a number of significant specific algorithms. These include distributed snapshots [ ], conflict resolution [ ], detection of quiescent properties [ ], distributed search [ ], etc. In addition, we pioneered some proof techniques for distributed systems [ ].

As the field matured and our understanding of the basic issues deepened, we looked into a unifying framework for studying *all* parallel programming problems. The scientific benefit in unifying diverse areas is obvious: research in one area carries over to the others, the issues that are fundamental to all areas—such as synchronization, mutual exclusion—can be studied and solved in their full generality and the issues of implementation on specific architectures can be separated from the scientific issues of problem solving. The potential economic benefits are attractive: if problem solving can be divorced from architectural considerations then a solution can be developed independent of the architecture and then implemented on specific architectures with modest amounts of effort.

Our work during 1985-1986 concentrated on developing such a unifying framework under the name UNITY. This work has attracted considerable attention—in a recent symposium on Concurrent Programming organized by C.A.R. Hoare (under the auspices of the Year of Programming at the University of Texas), UNITY represented a major share of the presentations and was featured largely during the discussions. We are in the process of writing a book (**Parallel Program Design: A Foundation**), to be published by Addison-Wesley later this year.

Our work in UNITY was to propose a simple model of computation and a logic to reason about properties of such programs. The simplicity of the model—a program is a set of assignment statements, whose execution consists of executing any arbitrary statement at each step with the restriction that every statement gets executed infinitely often—and the logic—the usual assertional style logic for sequential programs, proposed by Hoare, augmented with two temporal operators to reason about nonterminating executions—is striking. In fact, by restricting ourselves to a simple model we have managed to solve a number of thorny theoretical issues (dealing with fairness, for instance).

Our success with UNITY far exceeds our initial expectations. We have managed to study problems from a variety of problem areas—combinatorics, graph algorithms, matrix problems, operating system problems, communication protocols, fault tolerance, circuit design, and show that our unified model of computing yields simpler *and* more general solutions. We have developed a number of transformations which are appropriate for implementations on a variety of architectures: sequential, asynchronous shared memory, distributed message passing, synchronous parallel with shared memory, systolic arrays and VLSI chips. The diversity of the application areas and the architectures studied lends credence to our hypothesis that there is a UNITY to programming.

In the next section, we give a more detailed overview of UNITY and the current status of research. Among the other work supported by the AFOSR grant is a paper on Distributed Discrete Event Simulation which appeared in the Computing Surveys. This work has excited enough interest that commercial packages for distributed simulation are now available for the INTEL hypercube. We are told that a 128-processor hypercube delivers an 80 to 100 fold performance improvement over a VAX-750, using our distributed simulation algorithm.

THE UNITY OF THE PROGRAMMING TASK

Our work is about parallel programs; however, it is primarily about programs and secondarily about parallelism. The diversity of architectures and consequent programming constructs (send and receive, await, fork and join...) must be placed in the proper perspective with respect to the unity of the programming task. By stressing the differences, we are in danger of losing sight of the similarities. Our central thesis is that the unity of the programming task is of primary importance; the diversity is secondary.

The basic problem in programming is managing complexity. We cannot address that problem as long as we lump together concerns about the core problem to be solved, the language in which the program is to be written, and the hardware on which the program is to execute. Program development should begin by focusing attention on the problem to be solved and postponing considerations of architecture and language constructs.

Some argue that in cases where language and hardware are specified as part of a problem, concerns about the core problem, language, and hardware are inseparable. For instance, programs executing on a distributed network of computers must employ some form of message passing; in such cases concerns about message passing appear inseparable from concerns about the core problem. Similarly, since the presence or absence of primitives for process creation and termination in the programming language influence the program, it appears that language issues are inseparable from others. Despite these arguments, we maintain that it is not only possible but important to separate these concerns; indeed it is even more important to do so for parallel programs because parallel programs are less well understood.

Twenty-five years ago, many programs were designed to make optimum use of some specific feature of the hardware. Programs were written to exploit a particular machine language command or the number of bits in a computer word. Now, we know that such optimizations are best left to the last stages of program design or left out altogether. Today, parallel programs are designed much like sequential programs were designed in the 1950's: to exploit the message passing primitives of a language or the network interconnection structure of an architecture. A quarter century of experience tells us that such optimizations are *best postponed* until the very end of program development. We now know that a physicist who wishes to use the computer to study some phenomenon in plasma physics, for instance, should not begin by asking whether communicating sequential processes or shared-memory is to be used, any more

than whether the word size is 32 or 60 bits. Such questions have their place, but concerns must be separated. The first concern is to design a solution to the problem; the later concern is to implement the solution in a given language or for a particular architecture. Issues of performance on a specific architecture should be considered, but only at the appropriate time.

Programs outlive the architectures for which they were designed initially. A program designed for one machine will be called upon to execute efficiently on quite dissimilar architectures. If program designs are tightly coupled to the machines of today, program modifications for future architectures will be expensive. Experience suggests that we should anticipate requests to modify our programs to keep pace with modifications in architecture—witness attempts to parallelize sequential programs. It is prudent to design a program for a flexible abstract model of a computer with the intent of tailoring the program to suit future architectures.

An approach to exploiting new architectural features is to add features to the computational model. However, a baroque abstract model of a computer only adds to the complexity of programming. On the other hand, simple models such as the Turing Machine do not provide the expressive power needed for program development. What we desire is a model that is simple and has the expressive power necessary to permit the refinement of programs to suit target architectures.

The emphasis on the unity of the programming task is a departure from the current view of programming. Currently, programming is fragmented into subdisciplines, one for each architectural form. Asynchronous distributed computing, in which component processes interact by messages, is considered irrelevant to synchronous parallel computing. Systolic arrays are viewed as hardware devices and, hence, traditional ideas of program development are deemed inapplicable to their design.

Our goal is to show how programs may be developed in a systematic manner for a variety of architectures and applications. A criticism of this work is that its fundamental premise is wrong because programmers should *not* be concerned with architecture—compilers should. Some styles of programming—e.g., functional and logic programming–are preferred precisely because architecture is not their concern. Our response to this criticism is two-fold. First, programmers who are not concerned with architecture should not have to concern themselves with it—they should stop early in the program development process with a program which may or may not map efficiently to the target architecture. Second, there are some problems in which programmers have to be concerned with architecture either because the problem specifies the architecture (e.g., design a distributed command and control system) or because performance is critical; for these problems the refinement process is continued until efficient programs for the target architectures are obtained.

UNITY

We introduce a theory—a programming notation and proof system—called UNITY.

We choose to view our programs as Unbounded Nondeterministic Iterative Transformations—hence the term UNITY. In the interests of brevity, the phrase "a UNITY program" is preferred to "a program in unbounded nondeterministic iterative transformation notation." We are not proposing a programming language. We adopt the minimum notational machinery to illustrate our ideas about programming.

A UNITY Program

A program consists of a declaration of its variables and their initial values and a set of multiple assignment statements. Computation proceeds by executing any assignment statement selected nondeterministically. Nondeterministic selection is constrained by the following "fairness" rule: every statement is selected infinitely often.

The state of a program is given by the values of its variables. A *fixed-point* of a program is a state in which, for all assignments in the program: values on the left and right sides of the assignment are equal. A program execution does not terminate; if, however, a program is at a fixed-point then continued execution leaves the state unchanged, and in this sense, continued execution at a fixed-point has the same effect as ceasing execution.

Representing Programs for Various Architectures in Our Computational Model

Our model of programs is simple; in fact it may appear too simple for effective programming. We find that our model is adequate for the development of programs in general and parallel programs in particular. Now, we give an informal and very incomplete description of how different kinds of programs are represented in our model.

Our computational model is a nondeterministic state transition system. The state of a program is given by the values of its variables. A state change is effected by assigning values to one or more variables by executing a multiple assignment statement.

A synchronous system is one in which there is a global clock variable that is incremented with every state change. Multiple assignments model parallel synchronous operations.

A statement of the form **await** B **do** S in an asynchronous shared-variable program is encoded as a statement in our model which does not change the value of any variable if B is $false$ and otherwise has the same effect as S. A Petri net, another form of asynchronous system [ ], can be represented by a program in which a variable corresponds to a place, the value of a variable is the number of markers in the corresponding place, and a statement corresponds to a transition. The execution of a statement decreases values of variables corresponding to its input places by 1 (provided they are all positive) and increases values of variables corresponding to its output places by 1 in one multiple assignment.

Asynchronous message-passing systems with first-in-first-out error-free channels may be represented by encoding each channel as a variable whose value is a sequence of

messages (representing the sequence of messages in transit along the channel). Sending a message is equivalent to appending the message to the end of the sequence and receiving a message to deleting its head.

We cannot control the sequence in which statements are executed. However, by introducing variables (and using them in conditional expressions) we can ensure that the execution of a statement has no effect (i.e., does not change the program state) unless the statement execution occurs in a desired sequence.

Programs and Mappings

The UNITY notation is a mathematical notation to describe unbounded nondeterministic iterative transformations of the state of a system. A UNITY program describes *what* should be done in the sense that it specifies what the initial state and the state transformations (i.e., the assignments) are. A UNITY program does not specify precisely *when* an assignment should be executed—the only restriction is a rather weak fairness constraint: every assignment is executed infinitely often. Nor does a UNITY program specify *where*, i.e., on which processor in a multiprocessor system, an assignment is to be executed, or to which process an assignment belongs. Also, a UNITY program does not specify *how* assignments are to be executed or *how* fixed-points are detected in an implementation.

UNITY separates concerns between *what* on the one hand and *when, where* and *how* on the other. The *what* is specified in a program, whereas the *when, where* and *how* are specified in a mapping. By separating concerns in this way, a simple programming notation is obtained that is appropriate for a wide variety of architectures. Of course, this simplicity is achieved at the expense of making mappings immensely more important and more complex than they are now. This separation of concerns is a point of departure of UNITY from the imperative programming style.

Though the mappings described here are from UNITY to architectures, the mappings could just as well be to programming languages—for instance mappings could be proposed to PASCAL, CSP, and MODULA.

Mappings are illustrated here by proposing one from UNITY to von Neumann machines. The goal of this illustration is to give the reader some idea of what mappings are.

A mapping to a von Neumann machine specifies the schedule for executing assignments, the manner in which fixed-points are detected, and the manner in which multiple assignments are executed. We propose a mapping in which an execution schedule is represented by a finite sequential list of assignments in which each assignment in the program appears at least once. The computer executes this list of assignments repeatedly. The list is executed infinitely often (or equivalently, until a fixed-point is reached). We are obliged to prove that the schedule is fair, i.e., that every assignment in the program is executed infinitely often. Since every assignment in the program appears at least once in the list, and since the list is executed infinitely often, it follows

that every assignment in the program is executed infinitely often.

The implementation of multiple assignments on sequential machines is straightforward and is not discussed here.

To evaluate the efficiency of a program executed according to a given mapping, it is necessary to describe the mapping—the data structures and the computational steps—in detail. We shall not do so here because our goal is merely to emphasize the separation of concerns: programs are concerned with *what* is to be done whereas mappings are concerned with the implementation details of *where, when,* and *how.*

A description of a mapping from a UNITY program to a multiprocessing computer includes a specification of which processor is to execute a given assignment. There are implementations in which assignments—considered to be separate tasks—migrate from processor to processor. However, we restrict attention to a *static* allocation of assignments to processors. Thus the problem is one of partitioning the set of assignments of a UNITY program among a static set of processors. A mapping to a multiprocessing computer also specifies where the variables of a program are to be stored. In some architectures all variables reside in a common shared memory. In other architectures, each processor has its own local memory; it is more expensive for one processor to access a variable in another processor's memory than to access a variable in its own local memory—for such architectures the problem is that of partitioning variables and assignments among processors.

Descriptions of architectures and mappings can be made extremely detailed. Memory caches, I/O devices and controllers should be described if it is necessary to evaluate efficiency at that level of detail.

SUMMARY

We propose a unifying theory for the development of programs for a variety of architectures and applications. The computational model is unbounded nondeterministic iterative transformations of the program state. Transformations of the program state are represented by multiple assignments. The theory attempts to decouple the programmer's thinking about a program and its implementation on an architecture; we attempt to separate the concerns of *what* from those of *where*, *when*, and *how*. Details about implementations are considered in mappings of programs to architectures. We hope to demonstrate that we can develop, specify and refine solution strategies independent of architectures.

The utility of a new approach is suspect, especially when it is a radical departure from the conventional. Therefore, we have madea conscientious effort to apply our ideas to a number of architectures and application domains. Our experience, while not conclusive, is encouraging.

LIST OF PUBLICATIONS

[1] (with K. Mani Chandy) "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," *ACM Transaction on Programming Languages and Systems*, Vol. 8, No. 3, July 1986, 326-343.

[2] "Distributed Discrete-Event Simulation," *Computing Surveys*, Vol. 18, No. 1, March 1986

LIST OF PROFESSIONAL PERSONNEL

Name	K. Mani Chandy
Title	Co-Principal Investigator
Department	Faculty, Department of Computer Sciences, UT

Name	Jayadev Misra
Title	Co-Principal Investigator
Department	Faculty, Department of Computer Sciences, UT

INTERACTIONS (Invited Lectures Given on Topics Related to Wrok Performed Under This Grant)

- University of California at San Diego
  November 25, 1985
- Computer Measurement Group XVI Conference, Dallas
  December 10, 1985
- California Institute of Technology, Los Angeles
  March 4, 1986
- Microelectronics and Computer Technology Corporation (MCC)
  Spring 1986, 3-lecture series
- Lake Arrowhead (CA) Workshop on High Performance Computing
  September 9, 1986
- M.I.T. Workshop on Distributed Algorithms in Communication and Computation, Cambr
  October 23, 1986
- Workshop on Design and Implementation of Concurrent Programs, The Netherlands
  November 18-20, 1986

ADVISORY FUNCTIONS

Dr. Chandy serves on the Committee on Recommendations for U.S. Army Basic Scientific Research, National Research Council, July 1, 1984 to June 30, 1987.

END

Feb.

1988

DTIC